RETO ACCENTURE

Algoritmos de Optimización

No todo el monte es orégano

Marcos Brian Leiva Cerna
Eduardo Rivero Rodríguez
Pablo Villalobos Sánchez
Alberto Maurel Serrano

Introducción

Somos un equipo de 4º del Doble Grado de Matemáticas e Ingeniería Informática. Para resolver este problema, quizás el más algorítmico de todos los del concurso, hemos intentado combinar las técnicas matemáticas y capacidad crítica adquirida en el Grado de Matemáticas junto a los conocimientos algorítmicos obtenidos en el Grado de Ingeniería Informática.

En primer lugar, se presentan los pasos que hemos seguido para resolver el problema, junto a los algoritmos que hemos estudiado. A continuación, incluimos los conocimientos adquiridos y las dificultades encontradas. Por último, la bibliografía empleada y un anexo.

Metodología y estrategia desarrollada para la resolución del problema

Estamos ante un problema en el que el espacio de búsqueda crece de forma exponencial. Esto quiere decir que no podemos explorar todo el árbol de soluciones empleando algoritmos de fuerza bruta o búsqueda ciega, sino que tenemos que emplear algoritmos que recorran las partes más prometedoras del espacio de búsqueda.

Esto nos deja una gran cantidad de algoritmos que podemos probar: algoritmos heurísticos, metaheurísticos, ramificación y poda, … incluyendo todas sus variantes.

La aproximación que hemos seguido para resolver el problema ha sido la siguiente:

1. Observar las restricciones del problema.
2. Proponer algoritmos cuyo paradigma de funcionamiento se ajuste a estas restricciones.
3. Probar los algoritmos, ver los resultados y obtener feedback para mejorar los algoritmos actuales o probar otros nuevos.

1. Observar las restricciones que nos propone el problema

A lo largo de la resolución del problema nos hemos planteado las siguientes preguntas:

¿Es la división útil?

Intuitivamente, la división es una operación que no nos va a interesar utilizar. En primer lugar, tenemos que construir un número lo suficientemente grande como para poder dividirlo y obtener el nuestro. Después, no podemos dividir directamente, puesto que el número que hemos obtenido es un producto de primos (al dividir obtendríamos el producto de los otros primos que hemos empleado), por lo que tenemos que aplicar una suma o una resta en el proceso, añadiendo otra operación. Por último, tenemos que dividir, lo que nos lleva otra operación más, es decir, hemos empleado ya 7 operaciones para llegar a números del orden de 10^9.

Hemos querido comprobar si estas intuiciones eran correctas mediante pruebas experimentales. Construyendo hasta los números formados por hasta 4 primos, obtenemos que, excluyendo cualquier primo, no hay ni un solo número que podamos obtener empleando menos primos si utilizamos la división. Además, en la mayoría de casos la división no aparece en ni una sola de las formas de construir un número de forma óptima, y en los casos en los que lo hace hay otra forma de hacerlo en el mismo número de pasos sin utilizar la división.

A raíz de estas observaciones hemos decidido en un principio prescindir de la división e invertir el tiempo en el resto de operaciones, mucho más prometedoras.

¿Hasta qué punto podemos construir soluciones óptimas?

A la hora de resolver el problema la limitación va a estar evidentemente en el tiempo, y presuponemos que no se espera que hallemos soluciones óptimas, sino simplemente soluciones lo suficientemente buenas.

Esta pregunta es más interesante si lo que nos planteamos es obtener soluciones óptimas para posteriormente analizarlas e incorporar estas conclusiones a la construcción de nuestro algoritmo. Los datos obtenidos se describen y analizan en la sección: *Búsqueda en anchura*.

¿Qué datos podemos extraer de la observación de las soluciones óptimas en casos pequeños?

Queríamos acotar la estructura de las soluciones óptimas, con preguntas como:

- ¿Habrá operaciones que se utilicen más que otras?
- ¿En qué contexto se utiliza cada operación?
- ¿Para cada número cuántas soluciones óptimas habrá en promedio?

De aquí extrajimos varias conclusiones interesantes:

Para cada número no hay muchas formas de obtenerlo con el mínimo número de pasos posibles. Si eliminamos las que aparecen repetidas (como multiplicando los mismos números en otro orden) obtenemos que en la gran mayoría de los números se pueden obtener de 1 o 2 formas solamente, y que cuanto más grande sea un número en el nivel[1], menos formas hay de obtenerlo (puesto que como el número es muy grande para los pocos primos que podemos utilizar, tenemos poco margen de maniobra y no podemos "desperdiciar" operaciones).

Dependiendo del número mínimo de operaciones que necesitemos para obtener un número, la probabilidad de la última operación va a variar de forma sustancial. Esto además no va a variar prácticamente con el primo excluido. Los porcentajes obtenidos son aproximadamente:

	Nivel 2	Nivel 3	Nivel 4	Nivel 5
Sumas	0.38	0.347	0.278	0.285
Multiplicaciones	0.215	0.36	0.465	0.42
Restas	0.40	0.295	0.25	0.29

- ¿Podemos extrapolar estos porcentajes a niveles superiores?

La respuesta para nosotros es no, al menos con esta cantidad tan escasa de datos. Vemos que en los niveles 2, 3 y 4 la tendencia es que la multiplicación aumente su probabilidad y que la de las otras operaciones se reduzca, pero en el nivel 5 se modifica esa tendencia.

Por otro lado, esto es un promedio: para cada número los porcentajes variarán mucho, y más si tenemos en cuenta que para muchos de ellos hay una única forma de conseguirlos de forma óptima (es decir, si queremos obtener dicho número de forma óptima la probabilidad de una operación será de 1 y la de las restantes será 0).

[1] En el apartado búsqueda en anchura se define el significado de **nivel** de un número

¿Cómo es la estructura de las soluciones?

Esta pregunta ha sido la pregunta clave para nuestra solución, y su respuesta se desarrolla en el apartado de *Algoritmo voraz.*

2 y 3. Algoritmos que hemos propuesto y resultados que hemos obtenido

Primero haremos una descripción del algoritmo, y después recogeremos las ventajas y problemas que presenta. Luego indicaremos como hemos resuelto los problemas (en caso de haber podido resolverlos) y por último analizaremos los resultados obtenidos (en caso de haberlo implementado).

Búsqueda en anchura

Anteriormente ya hemos dicho que no podemos calcular las soluciones de forma óptima para números del orden de 10^9. Sin embargo, sí que podemos hacerlo para números más pequeños y después construir nuestros algoritmos sobre estos precálculos. Como esta es una técnica que vamos a emplear mucho (a pesar de no ser un algoritmo para resolver el problema original) consideramos que tiene una gran importancia y por eso le dedicamos este espacio.

Definimos **nivel de un número** como el mínimo número de primos[2] de una o dos cifras permitidos que necesitamos para obtener el número. La búsqueda en anchura primero va a calcular primero los números de nivel 1, que serán los primos permitidos.

Los números de nivel 2 surgirán de operar 2 números de nivel 1.

Los números de nivel 3 surgirán de operar un número de nivel 1 y otro de nivel 2.

Y en general, los números de nivel *n* serán aquellos formados por un número de nivel *i* y otro de nivel *n – i*, con i entre 1 y *n/2.*

La información relevante del desempeño del algoritmo está recogida en la siguiente tabla[3]:

	Nivel 2	Nivel 3	Nivel 4	Nivel 5
Tiempo (s)	0,007 – 0,008	0,145 - 0,155	4,3 - 5	165 - 200
Podemos obtener todos los números del 0 al	26 - 100	265 - 270	9985 - 11450	122000 - 142000
Números diferentes que podemos obtener	390 - 423	6500 -7250	124750 - 145000	$2,4*10^6$ – $2,7*10^6$

Esto son valores aproximados, y la oscilación se debe al primo en particular que decidamos excluir. Tenemos por tanto que en nuestro programa podemos calcular los niveles 2 y 3, pero si nos fijamos vemos que también podemos precalcular el nivel 4. Para ello vamos a resolver juntos todos aquellos casos que excluyan el mismo primo, ya que los números precalculados para todos ellos serán los mismos.

[2] A lo largo del trabajo nos referiremos a los números permitidos (el 1 y los primos de 1 y 2 cifras) simplemente como primos.

[3] Testado en C++ con el equipo de referencia indicado al final del documento. En la práctica en Python estos cálculos son mucho más rápidos, tardando en torno a medio segundo para el nivel 4.

Divide y vencerás probabilista

Algoritmo:

Pensamos ahora en abordar el problema en el sentido inverso: en vez de comenzar en el 0 e ir operando hasta llegar al número, vamos a partir del número y a dividirlo en números más pequeños sobre los que repetiremos el algoritmo, y así hasta llegar a los primos que podemos utilizar.

Sin embargo, no podemos probar todas las divisiones posibles en cada paso. Tenemos por tanto que escoger de alguna forma cómo partir el número en dos. Para realizar la elección vamos a emplear una mezcla de aleatoriedad y heurística:

- **Heurística** porque la probabilidad de escoger la forma de partir el número va a seguir unos ciertos criterios proporcionados por nosotros.
- **Aleatoriedad** porque un valor aleatorio será el que determine finalmente que operación realizar (de aquí que sea un algoritmo *probabilista*).

Lo ideal sería además que en cada paso probásemos varias formas de partir el número con el que estamos trabajando, y que después escogiésemos la forma que tuviese menor coste. Sin embargo, este algoritmo tiene una complejidad temporal en $O(r^p)$, donde r es el factor de ramificación y p es la profundidad máxima. Si en cada paso solo probamos una forma de dividir el número, el factor de ramificación es 2: cada vez que hacemos un paso tenemos que resolver el problema para 2 números. En concreto, si para cada número probamos k formas de partirlo diferentes, el factor de ramificación será $r = 2*k$, y comprobamos experimentalmente que con $k = 2$ el algoritmo ya tiene un tiempo de ejecución superior al permitido.

Realizaremos varias iteraciones, ejecutando en cada una de ellas el algoritmo, y nos quedaremos con la mejor solución encontrada.

Ventajas
Podemos realizar cualquier secuencia de operaciones hasta llegar al número.
Problemas
No disponemos de datos suficientes para extrapolar la función heurística.
Estamos decidiendo que operación realizar suponiendo que estamos en un cierto nivel, pero no tenemos forma de saber en qué nivel estamos.
Una vez decidida la operación, tampoco queda claro cómo debemos realizar la partición.
No está claro que el método sea convergente.

Resolución de problemas:

Los tres primeros problemas están relacionados con la función heurística. Como ya comentamos anteriormente no tenemos datos suficientes para extrapolar la función heurística. Por ello la probabilidad con la que realizamos las operaciones se va a basar en los resultados observados[4] y ver con qué valores obtenemos mejores resultados. Además, estamos ligando la heurística a un valor que no conocemos, el nivel del número con el que estamos trabajando. Esto tiene dos soluciones:

1. Emplear la misma probabilidad para cada una de las operaciones en todos los niveles. De esta forma no necesitaremos conocer en qué nivel estamos.

[4] En el punto *¿Qué datos podemos extraer de la observación de las soluciones óptimas en casos pequeños?*

2. Estimar en qué nivel estamos en base a soluciones que ya hemos encontrado previamente.

En cuanto a la partición que realizamos, si estamos trabajando con la multiplicación deberemos escoger uno de los divisores del número y partirlo en él y el cociente del número y el divisor. Parece razonable partirlo en dos números lo más similares posibles, para que así ambos sean lo menores posible y podamos llegar a ellos con un menor número de operaciones.

Si estamos trabajando con la suma o la resta lo lógico es no partir el número en dos números que sean muy similares, sino partirlo en uno pequeño que se pueda conseguir con una o dos operaciones y otro grande, pero al que podamos llegar rápido empleando multiplicaciones. Por las observaciones realizadas, en la mayoría de los casos excluyendo un primo podemos formar todos los números hasta el 100 con solo dos primos iniciales, por lo que parece razonable elegir en la suma o resta que el menor número esté en el rango [1,100]. Aun así, nos quedan bastantes candidatos en todas las operaciones, por lo que entre los candidatos tomaremos uno de forma aleatoria.

Por último, para garantizar la convergencia del método necesitamos poder asegurar que, aunque en ciertos momentos el número al partirlo se divida en un número más grande que el que teníamos (al realizar una resta dividiremos n en $n+m$ y m), tras realizar suficientes pasos los números van a decrecer. Sin embargo, estamos acotando el rango en el que el número puede crecer y además vamos a tener muchas más multiplicaciones y sumas que restas (las probabilidades de la multiplicación y suma van a ser mayores). Gracias a esto vemos experimentalmente que el algoritmo sí que es convergente. Lo que no podremos garantizar es la velocidad de convergencia del método.

Optimizaciones:

Una optimización que podemos hacer es precalcular los números que podemos obtener hasta un cierto nivel. Antes de partir un número, comprobaremos en nuestros precálculos si está el número, y si ya está no solo no necesitamos realizar más cálculos, sino que además tenemos garantizado que estamos consiguiendo el número en el menor número de pasos posible.

Resultados:

Nos hemos decantado por utilizar la misma probabilidad para todos los niveles, e ir variando esta probabilidad para ver si obtenemos alguna diferencia en los resultados. Realizando 1000 iteraciones por número, con las sumas y restas en el rango [1,100] sobre el caso de prueba de Accenture obtenemos los siguientes resultados:

Probabilidad de la suma	0.25	0,275	0,3	0,325	0,35
Probabilidad de la multiplicación	0.5	0,45	0,4	0,35	0,3
Probabilidad de la división	0.25	0,275	0,3	0,325	0,35
Media de primos necesarios	7,17	7,13	7,2	7.14	7.24

Por último, para el caso más prometedor (el segundo), hemos dejado 10.000 iteraciones para ver si el algoritmo sería capaz de mejorar sus resultados con más tiempo, y obtenemos que necesitamos una media de 6,92 primos por caso, lo cual es una notable mejora. Sin embargo, necesitamos entre 8 y 10 segundos por caso, sin incluir los precálculos, por lo que es totalmente inviable hacer tantas iteraciones por número.

Conclusiones:

Si bien es cierto que como punto de partida no obtenemos unos malos resultados, sí que son unos resultados bastante pobres comparados con los que obtendremos más adelante. Además, el tiempo de ejecución con 1000 iteraciones ronda los 200 segundos, por lo que no tenemos margen para que el algoritmo haga más iteraciones. El único margen de mejora que podríamos extraer sería mejorando la forma de partir el número, tanto el operador como el rango en el que los escogemos.

Modificando el rango de sumas y restas no obtenemos diferencias significativas, e intentando hacer evolucionar la heurística tampoco hemos conseguido mejores resultados. Es por eso por lo que hemos decidido desechar el algoritmo y probar con otros más prometedores.

A*

Algoritmo:

El A* es un algoritmo de búsqueda heurística. Va a explorar el espacio de soluciones, expandiendo en cada momento el nodo más prometedor, atendiendo a una heurística que le hayamos indicado.

Ventajas
Si obtenemos una buena heurística tendremos soluciones parcialmente óptimas
Explora solo las ramas más prometedoras, siendo mucho más rápido que la búsqueda ciega.
Problemas
No está claro cómo construir las soluciones
Tenemos que encontrar una buena función heurística

Resolución de problemas:

El primer problema con el que nos encontramos es que en el algoritmo A* tenemos que ir construyendo la solución paso a paso, y si utilizamos como operador para avanzar la suma, resta, multiplicación y división con un número primo, entonces todos los números que formemos van a tener la forma:

$$n = (((p_1 \blacksquare p_2) \blacksquare p_3) \blacksquare p_4) \blacksquare \ldots$$

donde ■ es una operación de las permitidas y p_i un primo. Si hacemos esto, va a ser extremadamente difícil llegar a los números grandes: primero multiplicaremos varios números grandes hasta llegar a uno "relativamente cercano" y después tendremos que ajustarlo al número que queremos empleando sumas y restas de números de 2 cifras (cuando estamos trabajando con números de 9 cifras es bastante complicado). Esto último se puede paliar si primero realizamos una búsqueda en anchura hasta el nivel 3 o 4 y posteriormente para cada estado al que lleguemos preguntamos si realizando una operación entre el número del estado y uno de los números precalculados podemos llegar al objetivo.
El segundo problema que tenemos es encontrar una buena heurística. Esto tiene una doble importancia:

1. Cuanto más informada sea la heurística, llegaremos antes al resultado y exploraremos las ramas más prometedoras del árbol de estados.
2. El A* lo implementaremos con control de repetidos. Por tanto, si la heurística es consistente[5], garantizaremos la optimalidad del resultado (el resultado sería óptimo dentro del problema restringido a solo usar las operaciones de esta forma, el resultado obtenido en general no será óptimo si podemos poner paréntesis de cualquier forma, por eso las llamamos parcialmente óptimas).

[5] Una heurística es consistente sí solo sí para todos los n_i y n_j (siendo n_j un estado adyacente a n_i) se cumple: $h'(n_i) - h'(n_j) \leq coste(n_i,n_j)$, siendo $h'(x)$ el valor de la heurística en el nodo x y coste(x,y) = coste para ir de x a y.

Probamos con las siguientes heurísticas:

```
Heurística1(estado) = abs(estado.número - númeroObjetivo)
```

Empezamos con la heurística que a cualquiera se le ocurriría, explorar antes los estados que están más cerca del objetivo. Con esta heurística en realidad no estamos empleando el algoritmo A*, sino haciendo una búsqueda voraz.

```
Heurística2(estado) = max(estado.número, númeroObjetivo) /
                              min(estado.número, númeroObjetivo) + 100 * estado.pasos
```

En la heurística 2 damos mucha prioridad al número de pasos realizados, siempre y cuando no estemos extremadamente lejos del número objetivo.

```
Heurística3(estado) = max(estado.número, númeroObjetivo) /
                                 min(estado.número, númeroObjetivo) + estado.pasos
```

Por último, con esta heurística si estamos lejos del número objetivo primamos la distancia y si estamos cerca primamos el número de pasos realizados (cuántos primos hemos usado en nuestra solución parcial). Es por tanto un híbrido de las dos ideas anteriores.

Resultados:

Sobre el archivo de prueba proporcionado por Accenture obtenemos los siguientes resultados:

	Heurística 1	Heurística 2	Heurística 3	Mejores soluciones[6]
Números necesarios	8 - 19	8 - 10	8 - 11	6 - 8
Tiempo ejecución por caso (s)	30	14	18 - 22	

Dado los buenos resultados que hemos obtenido con la heurística 2, nos hemos animado a probar más heurísticas de la forma

```
Heurística(estado) = max(estado.número, númeroObjetivo) /
                              min(estado.número, númeroObjetivo) + k * estado.pasos
```

con *k* variable. Con $k \approx 600$ obtenemos los mejores resultados, manteniendo el tiempo de ejecución por caso en unos 14 segundos y con soluciones de entre 7 y 9, necesitando un promedio de 8,45 números.

Conclusiones:

Aunque hayamos encontrado heurísticas que logran unos resultados bastante mejores que los que obteníamos al principio, para todas las heurísticas obtenemos resultados peores que los del algoritmo anterior y tiempos de ejecución muy grandes. Entendemos que el problema no está en las heurísticas elegidas sino en la forma que tiene el algoritmo de construir la solución. Tendremos que estudiar por tanto algoritmos que permitan insertar los paréntesis en otros lugares más convenientes.

[6] Estas soluciones han sido encontradas con el algoritmo voraz que expondremos más adelante, y no tenemos garantía de que sean óptimas.

Algoritmo voraz

Dado que las aproximaciones anteriores empleando algoritmos más clásicos habían fallado (eran lentos y las soluciones no estaban cerca de ser óptimas) decidimos intentar replicar de forma voraz la construcción de las soluciones óptimas.

Algoritmo:

A lo largo del proyecto habíamos ido observando que la mayoría de las soluciones buenas lo que hacían era aproximarse muy rápidamente al número objetivo y después ajustarlo con números pequeños. Queríamos además permitir poner cualquier combinación de paréntesis. Hicimos entonces la siguiente hipótesis: vamos a suponer que las soluciones son de la forma:

$$objetivo = a \cdot b \pm c$$

Intuitivamente lo que estamos haciendo es acercarnos rápidamente al número objetivo con *a x* b y posteriormente ajustarlo con *c*. La inclusión del resto (*c*) es importante, porque como hemos observado en la búsqueda en anchura parece que al aumentar el nivel disminuye la probabilidad de utilizar la multiplicación en el último nivel. Por otro lado, el utilizar *a x b,* en vez de hacer directamente *objetivo = a + r* atiende a los datos extraídos del A*: escribir el número objetivo de la forma:

$$objetivo = ((((p_1 \blacksquare p_2) \blacksquare p_3) \blacksquare p_4) \blacksquare \dots) \pm r$$

con p_i primo no acaba de dar buenos resultados.

Sin embargo, probar con todos los valores de *a*, *b* y *c* requeriría una ingente cantidad de tiempo. Por tanto, vamos a explorar los valores de *a*, *b* y *c* de forma inteligente para perder la menor cantidad de soluciones posibles y lograr que el algoritmo consiga buenos resultados en un tiempo moderado.

Como sabemos que podemos construir las soluciones óptimas hasta el nivel 4 en un tiempo reducido vamos a aprovecharnos de ello. Permitimos que *a* y *b* sean números de hasta de nivel 4 y *c* de hasta nivel 2. Pero en vez de probar con todos los valores de nivel 4 para *a*, vamos a hacerlo solo desde 2 hasta $\sqrt{objetivo}$. De esta forma no perdemos ninguna solución (ya que *a* y *b* no pueden ser simultáneamente mayores a $\sqrt{objetivo}$[7]). Y en vez de mirar para cada uno de los valores de *b* vamos a mirar para cada uno de los posibles valores de resto (que como es de nivel a lo sumo 2, va a tener a haber como mucho unas 7250 posibilidades[8]).

Hemos reducido de las $145000 \times 145000 \times 7250 = 152 \times 10^{12}$ posibilidades a "tan solo" $\sqrt{10^9} \times 7250 = 2 \times 10^9$. Sin embargo, en realidad las posibilidades son aún menos, puesto que cuando *a* sea medianamente grande, tan solo podremos utilizar unos pocos restos de nivel 2:

$$a \times b + c = objetivo = a \times (b + 1) + c \Rightarrow c = a + r$$

Como *c* y *r* están acotados por 7250 los casos que hay que explorar son poco más de $2 \times \sqrt{10^9} \approx 63250$

Acabamos de reducir el tamaño del espacio de búsqueda a un tamaño sobre el que sí podemos buscar de forma ciega.

---

[7] En realidad, sí que podrían ser ligeramente superiores, si restamos un número grande

[8] Calculado en *Búsqueda en anchura*

Ventajas
Es extremadamente sencillo de implementar
Muy rápido
Problemas
La estructura de las soluciones es fija, y no tenemos ninguna garantía que sea mejor que otra estructura
Es difícilmente escalable

El que la estructura de las soluciones sea fija es un problema, porque puede que en casos particulares haya una forma mejor de llegar al número óptimo, y aunque no tengamos ninguna garantía de que es mejor que otros métodos, todas las anteriores observaciones nos dicen que debería de funcionar bastante bien.

Casuística:

Sin embargo, lo que tenemos hasta ahora son solo intuiciones. Lo que va a llevar a este algoritmo al siguiente nivel es estudiar de qué formas podemos construir un número. Queremos es demostrar que las soluciones óptimas o bien tienen esta estructura (a la que llamaremos canónica), o bien tienen estructuras muy concretas que podemos explorar de forma rápida independientemente. Supongamos que **la división no interviene en las soluciones óptimas**[9]. Tenemos entonces las siguientes posibilidades:

Nivel ≤ 4

Como construimos mediante una búsqueda en anchura hasta el nivel 4 de forma óptima, tenemos garantizado que el número estará en el marcaje y por tanto la solución que obtenemos es óptima.

Nivel 5

Supongamos ahora que tenemos un número que hemos construido con solo 5 primos. ¿Podríamos hacerlo solo con 4? La respuesta es no, ya que, si pudiésemos, habríamos obtenido en la búsqueda en anchura la solución óptima. Por tanto, el resultado obtenido es óptimo.

Nivel 6

Hagamos ahora lo mismo para el nivel 6. Si nuestro número tuviese una solución de nivel 5, esta tendría una de las siguientes formas:

Caso 1: $n_4 x\ n_1 = a\ x\ b + 0$
Caso 2: $n_3 x\ n_2 = a\ x\ b + 0$
Caso 3: $n_4 + n_1 = 1\ x\ b + c$
Caso 4: $n_3 + n_2 = 1\ x\ b + c$

Donde $\mathbf{n_i}$ **es un número cualquiera de nivel i**. Todos estos casos encajan con el esquema de la solución canónica ($a\ x\ b + c$, siendo a y b números de nivel menor o igual a 4 y c un número de nivel menor o igual a 2). Por tanto, las soluciones de nivel 6 también son óptimas.

Nivel 7

Suponiendo que podemos expresarlo como un número de nivel 6, podemos reducir todas las posibilidades de combinar los números a los siguientes 8 casos:

Caso 1: $n_4 x\ n_2 = a\ x\ b + 0$
Caso 2: $n_3 x\ n_3 = a\ x\ b + 0$
Caso 3: $n_4 x\ n_1 + n_1 = a\ x\ b \pm c$
Caso 4: $n_3 x\ n_2 + n_1 = a\ x\ b \pm c$
Caso 5: $1\ x\ n_4 + n_2 = a\ x\ b \pm c$
Caso A: $n_1\ x\ (n_4 + n_1)\ /\ n_1\ x\ (n_4 - n_1)$
Caso B: $n_1 x\ (n_3+n_2)\ /\ n_1 x\ (n_3-n_2)\ /\ n_1\ x\ (n_2-n_3)$
Caso C: $n_3 \pm n_3$

[9] Suposición bastante razonable por lo discutido en *¿Es la división útil?*

Los casos 1 – 5 están contenidos en la forma canónica. Nos queda tratar los casos A, B y C, que lo haremos de forma independiente en la función *opt7*. Por tanto, tratando estos casos por separado garantizamos de nuevo la optimalidad del número conseguido.

Los casos A y B se pueden reducir a iterar por los primos admitidos y después iterar por los números de nivel menor o igual a 2 (estaríamos fijando el primer y el tercer número y hallando el segundo). Esto tiene una complejidad temporal en $O(L_1 \times L_2)$, siendo L_i cuántos números hay en el nivel i. Experimentalmente sabemos que L1 = 25 [10] y $L_2 \leq 7250$, por lo que la complejidad es asumible.

El caso C por su parte se resuelve fijando uno de los números de nivel 3 y calculando el otro, por lo que su complejidad temporal está en $O(L_3)$ con $L_3 \leq 145.000$, por lo que de nuevo es una complejidad razonable.

Nivel 8

Repetimos el mismo procedimiento que en los niveles anteriores:

Caso 1: $n_4 \times n_3 = a \times b + 0$

Caso 2: $(n_4 \times n_2) + n_1 = a \times b + c$

Caso 3: $(n_3 \times n_3) + n_1 = a \times b + c$

Caso 4: $(n_4 \times n_1) + n_2 = a \times b + c$

Caso 5: $(n_3 \times n_2) + n_2 = a \times b + c$

Caso A: $(n_4 + n_1) \times n_2 / (n_4 - n_1) \times n_2$

Caso B: $(n_3 + n_2) \times n_2 / (n_3 - n_2) \times n_2 / (n_2 - n_3) \times n_2$

Caso C: $(n_1 \times (n_1 + n_4)) + n_1 / (n_1 \times (n_1 + n_4)) - n_1 / (n_1 \times (n_4 - n_1)) + n_1 / (n_1 \times (n_4 - n_1)) - n_1$

Caso D: $(n_1 \times (n_2 + n_3)) + n_1 / (n_1 \times (n_2 + n_3)) - n_1 / (n_1 \times (n_2 - n_3)) + n_1 / (n_1 \times (n_2 - n_3)) - n_1 / (n_1 \times (n_3 - n_2)) + n_1 / (n_1 \times (n_3 - n_2)) - n_1$

Caso E: $n_3 + n_4$

Los casos que encajan con la estructura canónica son del 1 al 5. De nuevo tendremos que comprobar aparte los casos A – E. El caso E es idéntico al caso C del nivel 7, e incluso con la implementación que hemos hecho podemos fusionar ambos. Los casos A y B se comprueban fijando n_1 y n_2 o n_2 y n_2 y buscando en el marcaje n_4 o n_3. Ambos se pueden fusionar y tienen una complejidad temporal en $O((L_2)^2)$, totalmente asumible. Por último, tenemos los casos C y D que se pueden resolver fijando los 3 números menores y buscando el cuarto, con una complejidad temporal en $O((L_1)^2 \times L_2)$. Tendremos de nuevo por tanto que las soluciones de nivel 8 también serán óptimas.

Nivel 9

Experimentalmente hemos visto que, tras aplicar las mejoras a los niveles anteriores, necesitábamos como máximo 9 primos para formar cualquier número, y que en particular hemos encontrado solo 14 números que requieren 9 primos en 1000 casos, es decir, representarían tan solo un 1,4 % de los casos[11]. El bajo porcentaje de aparición de estos casos unido a que la cantidad de posibles estructuras con 8 números es inmensa nos han llevado a aplicar las optimizaciones solo en aquellos casos en los que son más sencillas de realizar. Los casos que hemos optimizado han sido los siguientes:

Caso A: $n_2 \times n_3 \times n_3$

Caso B: $n_2 \times n_2 \times n_4$

Caso C: $n_3 \times (n_1 + n_4)$

Caso D: $n_3 \times (n_2 + n_3)$

Procediendo como en apartados anteriores vemos que todas ellas son sencillas de implementar. Tras probar las optimizaciones sobre el mismo caso tan solo 6 números necesitaban 9 primos, menos de un 1%.

[10] Tras quitar el primo excluido.

[11] Aunque es una muestra pequeña y puede haber grandes variaciones, las pruebas experimentales posteriores apoyan la teoría de que los números de nivel 9 son extremadamente raros.

Resultados:

Probamos el algoritmo junto a las diferentes optimizaciones para ver cuál es la mejoría que obtenemos con cada una de ellas[12]:

	Media de números necesarios	Tiempo total
Solución canónica	7,555	300,14
Solución canónica + Optimización nivel 7	7,547	299,17
Solución canónica + Optimizaciones niveles 7 y 8	7,455	319,60
Solución canónica + Optimizaciones niveles 7, 8 y 9	7,449	320,73

Vemos que la mejoría que introducen las optimizaciones es notable y que el coste adicional es pequeño, así que hemos decidido mantenerlas todas. Ahora probamos sobre varios ficheros de prueba de tamaño 100 para ver cuál será su rendimiento esperado durante su ejecución:

	Test Accenture	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5
Media números necesarios	6,62	7,31	7,41	7,44	7,46	7,42
Tiempo ejecución (s)	22,76	39,70	40,09	40,91	42,27	42,25

Por último, hemos testeado el algoritmo con un archivo con 10000 casos, obteniendo que tan solo 82 necesitaban 9 primos y que en promedio necesitábamos 7,42 primos.

Conclusiones:

Este algoritmo es el más rápido de los que hemos encontrado, y salvando la suposición de que no se puede utilizar la división, es prácticamente óptimo en todos los casos (de acuerdo con lo que hemos visto en los 10000 casos, un 99,17% de los casos nos devuelve la solución óptima). Hemos probado además varias formas de utilizar la división y ninguna de ellas nos ha aportado mejores soluciones, por lo que consideramos que descartar las divisiones no nos quita un número significativo de soluciones óptimas.

Solo encontramos dos problemas al algoritmo. El primero de ellos es que es poco escalable en cuanto a tamaño de los números. Con algo más de tiempo podríamos llegar a precalcular hasta el nivel 5 y eso nos permitiría tratar los números de hasta nivel 9 y 10 con relativa facilidad. Pero el precálculo tiene una complejidad temporal exponencial, por lo que no podemos ir mucho más allá. Sin embargo, el problema al que nos estamos enfrentando tiene esta restricción en cuanto el tamaño y nos hemos adaptado a ella. Quizás otros algoritmos que no se viesen tan afectados por el tamaño se viesen afectados por otras restricciones para las que este es bastante robusto. Además, hemos propuesto otros algoritmos que podríamos utilizar si quisiésemos escalarlo.

La segunda es que es muy *ad hoc* para el problema y las restricciones proporcionadas, y que puede parecer que es una simple fuerza bruta. Sin embargo, no creemos que sea así. Por un lado, consideramos que la acotación del espacio ha sido bastante cuidadosa, por lo que no estaríamos empleando fuerza bruta. Por otro lado, hemos tenido que ser bastante ingeniosos viendo como hacíamos los bucles para garantizar que minimizábamos la complejidad temporal a la vez que no perdíamos soluciones.

Este es el algoritmo que hemos acabado implementando en SCRIPT.py, de ahí que hayamos hecho un desarrollo más exhaustivo del mismo.

[12] Probado con 1000 números aleatorios de hasta 10^9 en Python

Ant Colony Optimization (ACO)

Por último, presentamos una aproximación que hemos estudiado bastante teóricamente y creemos que podría aportar un nuevo punto de vista. Sin embargo, la dificultad de ajustar el modelo al problema, así como el poco tiempo del que disponíamos nos llevó a implementar otros métodos más sencillos.

Algoritmo:

El algoritmo de optimización por colonia de hormigas es un algoritmo metaheurístico que trata de simular el mecanismo que emplean las hormigas para hallar el camino óptimo desde el hormiguero hasta una fuente de comida. Al andar, las hormigas liberan feromonas. Las hormigas que encuentran antes comida vuelven más rápido al hormiguero, reforzando las feromonas del camino que han recorrido (las feromonas de todos los caminos se van evaporando con el tiempo). Al haber mayor cantidad de feromonas en los caminos más cortos, las hormigas posteriores tendrán mayor probabilidad de escogerlos, retroalimentando el mecanismo.

En este problema en concreto, el hormiguero estaría en el 0 y la comida en el número objetivo. El grafo mediante el que representaremos tendrá como nodos los distintos números que podamos usar.

Ventajas
Prácticamente no hacemos acotaciones sobre la estructura de las soluciones
Problemas
La matriz de feromonas es imposible de almacenar por su tamaño
Si no queremos acotar la estructura de las soluciones el factor de ramificación es inmenso
El algoritmo está pensado para operadores unarios

Resolución de problemas:

Para resolver el problema del tamaño de la matriz de feromonas hemos buscado un paper en el que se hubiese estudiado este problema. Tuvimos bastante suerte y encontramos uno (Marek Behálek, 2015). Empleando una variante particular del ACO, la Max-Min, veían que en ella la matriz de feromonas es una matriz dispersa, y que en la mayoría de las posiciones tienen el mismo valor de feromonas (en cada casilla de la matriz se guardan las feromonas del camino que une los nodos *x* e *y*). Proponen además varias soluciones para almacenar la matriz.

Ya habíamos visto en el algoritmo A* la importancia de que las soluciones no se basen solo en ir añadiendo un número detrás del que ya llevamos calculado. Una idea para evitar esto es precalcular de forma óptima con la búsqueda en anchura hasta $\sqrt{n}$, siendo *n* el número objetivo. Posteriormente, el número que tenemos actualmente podríamos operarlo con cualquiera hasta el $\sqrt{n}$.

De esta forma, aunque la solución tenga estructura (en cada paso operamos el resultado con un operador y un número):

$$n = (((\mathrm{n}_1 \blacksquare \mathrm{n}_2) \blacksquare \mathrm{n}_3) \blacksquare \mathrm{n}_4) \blacksquare \ldots$$

estaríamos permitiendo que los n_i fuesen números de hasta nivel 4 y las soluciones virtualmente tuviesen cualquier estructura interna[13].

Por último, el algoritmo está pensado para operadores unarios. Por tanto, en cada paso tenemos que elegir uno de los 4 operadores y uno de los números hasta $\sqrt{n}$ con el que queremos operarlo. El factor de ramificación sería de $4\sqrt{n}$, algo totalmente intratable. Tendríamos que elegir de alguna forma qué parejas

[13] Salvo casos extremos en los que se sumasen números muy grandes, pero estos casos son poco probables, de ahí el *virtualmente* cualquier estructura.

(*operador*, *número*) queremos probar con el número actual, volviendo de nuevo a los problemas que teníamos con el algoritmo *divide y vencerás probabilista*.

Conclusiones:

A nivel teórico puede ser una aproximación interesante, especialmente implementando las soluciones propuestas en el paper junto a las del algoritmo *divide y vencerás probabilista*, pero seguimos teniendo una gran cantidad de problemas que nos hacen dudar de la viabilidad del modelo, y más cuando no hemos conseguido pulir el algoritmo de divide y vencerás.

Conocimientos adquiridos

Los conocimientos y habilidades que hemos adquirido y trabajado durante la resolución del problema han sido los siguientes:

- Iniciación a Python (nunca habíamos trabajado con este lenguaje, aunque sí que teníamos experiencia programando con otros lenguajes).
- Introducción a los algoritmos metaheurísticos, especialmente al algoritmo de optimización por colonia de hormigas.
- Análisis sistemático de un problema.
- Ajustar paradigmas algorítmicos conocidos a problemas concretos.

Principales problemas que hemos encontrado

A lo largo de la resolución del problema hemos encontrado varios problemas:

- No teníamos claro por qué algoritmo empezar a probar, y nos ha costado mucho encontrar un modelo que funcionase correctamente.
- A nivel técnico, hemos encontrado una gran diferencia entre los tiempos de ejecución en unas situaciones y otras, por lo que nos ha costado hacernos a la idea de cuánto tiempo tarda realmente nuestro algoritmo y de si iba a entrar en el tiempo proporcionado o no.
- Ha resultado difícil probar el algoritmo, ya que no sabíamos hasta qué punto nuestras soluciones eran buenas.

Fuentes consulta

Gran parte de los algoritmos que hemos utilizado los habíamos estudiado previamente en las asignaturas de algoritmia, por lo que no hemos necesitado referencias.

Sí que hemos empleado bibliografía para el algoritmo A*, para el que hemos utilizado el material de la asignatura de Inteligencia Artificial I, que lamentablemente no hemos encontrado disponible online.

Por último, sí que hemos investigado los algoritmos metaheurísticos, para los que hemos utilizado las siguientes publicaciones:

- Algoritmo de la colonia de hormigas - Wikipedia, la enciclopedia libre. (s.f.). Recuperado de https://es.wikipedia.org/wiki/Algoritmo_de_la_colonia_de_hormigas
- Ant Colony Optimization. (2018, 3 noviembre). Recuperado de https://www.baeldung.com/java-ant-colony-optimization

- Metaheuristic algorithms. (s.f.). Recuperado de https://en.wikipedia.org/wiki/Metaheuristic
- Behálek, M., Surkovský, M., Meca, O., & Böhm, S. (2015). Memory optimized pheromone structures for max-min ant system.. Neural Network World, 161–174. https://doi.org/10.14311/NNW.2015.25.008

Anexo: Equipo y metodología empleados en las pruebas

El equipo que hemos empleado para realizar las pruebas se compone de:

Procesador: Ryzen 5 3600, 6 Cores 12 Hilos, 3,6 GHz
Memoria RAM: 2*8 GB = 16 GB

Según los benchmarks que hemos consultado este procesador es un 30% más potente en cálculos mononúcleo que sobre el que se va a ejecutar el script. Pero dado que nuestro algoritmo tarda unos 40 segundos por cada 100 casos consideramos que hay margen de sobra para que no haya problemas con el tiempo.

Las pruebas de *algoritmo voraz* las hemos realizado en Python con la versión definitiva del algoritmo, que es el que hemos presentado en SCRIPT.py. El resto de pruebas que hemos realizado han sido en C++. Dado el escaso tiempo que teníamos para realizarlas nos hemos decantado por programarlas en el lenguaje que nos resultaba más familiar. Aparte de ver los resultados, la importancia de las pruebas era comparar el rendimiento de los diferentes algoritmos. Al estar todas las pruebas realizadas en el mismo lenguaje, los tiempos sí que son comparables, que era lo que buscábamos. En todas aquellas pruebas en las que sea relevante, están indicadas las condiciones particulares en las que se ha realizado.